University of California
Santa Cruz

Optimal scheduling for tasks with stochastic runtimes

A project submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Statistics and Applied Mathematics

by

Niranjan Vissa

June 2015

The Project of Niranjan Vissa

is approved:

_____

Professor David Draper

_____

2nd person

Contents

List of Tables

List of Figures

ACKNOWLEDGMENTS

This master's project could not have been written without the help of Prof. David Draper who gave me the guidance, encouragement, and expertise needed for the advancement of my education.

Chapter 1

Introduction

This is the introduction

1.1 Section 1

This is intro - section 1

1.2 Section 2

This is intro - section 2

This is a reference to (Lesaffre, Leman, and Martens 2006)

Chapter 2

Single Processor case

2.1 Methods

Methods

2.2 Results

Results

2.3 Discussion

Discussion

Chapter 3

Multiple Processor case

3.1 Methods

Methods

3.2 Results

Results

3.3 Discussion

Discussion

Chapter 4

Conclusions and future work

4.1 Conclusions

This is the conclusion

4.2 Future work

This is future work to be done

4.2.1 Extension to Spot instances

Currently, cost = runtime (hrs) x cost ($/hr). Cost is assumed to be fixed while runtime is variable. Spot instances are much cheaper by their cost is variable. This introduced additional level of uncertainty into the model. Cost for Spot instances is given as time series data generated from an unknown model. Need to model the cost effectively and pick a maximum bid price such that the job is not interrupted because the current Spot price exceeds the max bid price. Can extend this further by using a low bid price with checkpointing and re-processing the task that was interrupted and processing all remaining tasks Need to consider the possibility that might not get a Spot instance and will have to use an On Demand instance instead for the remaining tasks.

4.2.2 Variance of runtimes & trainingset sample sizes

Currently using a fixed number of samples for each task size. It is possible that different task sizes will have different number of training set samples. Runtimes for task sizes with large number of training set samples will have lower variance that runtimes for sizes with fewer number of samples. Also, the same task size can have different number of samples for different instances types. Similarly, runtimes for a size on an instance with a large number of samples will have lower variance than runtimes for the same size on other instances with fewer number of samples. Can choose to run task on instance

type with fewer number of samples in case it turns out to faster than an instance type with more samples. Possible use of multi-armed bandits to balance explore vs. exploit here since runtimes for current set of tasks will be added to training set for tasks from the next job.

4.2.3 Move bootstrap code from R to C++

All code is currently in R and can take several minutes to run depending on the number of instances and tasks. Move the bootstrap code to C++ and call from R via RCpp to improve runtimes.

4.2.4 Skip runtime estimation for cluster sizes that are very unlikely to complete the tasks by the deadline

When trying to determine the minimum number of instances in a cluster of a certain instance type that will complete the job by the deadline, we should ignore clusters with too few instances and focus only on cluster sizes that have a reasonable chance of completing the job by the deadline. Use the min values in the training set for each size and see if the cluster is able to complete them by the deadline. If not, move on to the next size.

Bibliography

Lesaffre, Micheline, Marc Leman, and Jean-Pierre Martens. 2006. "A User-Oriented Approach to Music Information Retrieval." In *Dagstuhl Seminar Proceedings*, edited by T. Crawford and R.C. Veltkamp, 1–11. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI), Schloss. http://drops.dagstuhl.de/vollt/.

Appendix A

Code for schedulr R package

This is the code for the schedulr R package

```
## @knitr all

# Functions for Simulated annealing


data.env <- new.env()

# If an instance has more than bootstrap.threshold tasks, use Normal approx.
# to get runtime dist., instead of generating bootstrap samples
bootstrap.threshold <- 50
num.bootstrap.reps <- 1000



# -----
# Internal functions for validating input
# -----


#' Validate that the input value is a positive integer (test single number, not arr
#'
#' @param val The value to validate
#' @examples
#' check.if.positive.integer()
#' check.if.positive.integer(c())
#' check.if.positive.integer('')
#' check.if.positive.integer(5)
#' check.if.positive.integer(0)
```

```r
#' check.if.positive.integer(-10)
#' check.if.positive.integer(3.14)
#' check.if.positive.integer(1:2)
#' check.if.positive.integer('a')
.check.if.positive.integer <- function(value) {

  .check.if.nonnegative.integer(value)
  value > 0 || stop("Invalid argument: Value must be > 0")

} # end function - .check.if.positive.integer


#' Validate that the input value is a non-negative integer (test single number, not
#'
#' @param val The value to validate
#' @examples
#' check.if.nonnegative.integer()
#' check.if.nonnegative.integer(c())
#' check.if.nonnegative.integer('')
#' check.if.nonnegative.integer(5)
#' check.if.nonnegative.integer(0)
#' check.if.nonnegative.integer(-10)
#' check.if.nonnegative.integer(3.14)
#' check.if.nonnegative.integer(1:2)
#' check.if.nonnegative.integer('a')
.check.if.nonnegative.integer <- function(value) {

  !missing(value) || stop("Missing required argument: Must specify a value")
  length(value) == 1 || stop("Invalid argument length: Must specify a single numbe
  (is.numeric(value) && value == floor(value)) || stop('Non-integer argument: value
  value >= 0 || stop("Invalid argument: Value must be >= 0")

} # end function - .check.if.nonnegative.integer


#' Verify that the input value is a positive real (test arrays)
#'
#' @param value Array of values to validate
#' @examples
#' .check.if.positive.real()
#' .check.if.positive.real(c())
#' .check.if.positive.real('')
```

```
#' .check.if.positive.real(0)
#' .check.if.positive.real(1)
#' .check.if.positive.real(3.14)
#' .check.if.positive.real(-5)
#' .check.if.positive.real(c(1.2, 3.4))
#' .check.if.positive.real('a')
.check.if.positive.real <- function(value) {

  .check.if.nonnegative.real(value)
  all(value > 0) || stop('Invalid argument: Value must be > 0')

} # end function - .check.if.positive.real


#' Verify that the input value is a non-negative real (test arrays)
#'
#' @param value Array of values to validate
#' @examples
#' .check.if.nonnegative.real()
#' .check.if.nonnegative.real(c())
#' .check.if.nonnegative.real('')
#' .check.if.nonnegative.real(0)
#' .check.if.nonnegative.real(1)
#' .check.if.nonnegative.real(c(1.2, 3.4))
#' .check.if.nonnegative.real(3.14)
#' .check.if.nonnegative.real('a')
.check.if.nonnegative.real <- function(value) {

  !missing(value) || stop('Missing required argument: Must specify a value')
  length(value) > 0 || stop('Invalid argument length: Must specify a value')
  is.numeric(value) || stop('Non-numeric argument: Must specify a valid +ve real nu
  all(value >= 0) || stop('Invalid argument: Value must be >= 0')

} # end function - .check.if.nonnegative.real


#' Verify that assignment is valid
#'
#' @param assignment Array of task sizes
#' @examples
#' a <- get.initial.assignment(2, 3)
#' .validate.assignment(a)
```

```
#' .validate.assignment(b<-NULL)
.validate.assignment <- function(assignment) {

  !missing(assignment) || stop("Missing required argument: assignment")
  is.list(assignment) || stop("Invalid argument type: assignment must be a list")
  length(assignment) != 0 || stop("Invalid argument length: assignment must contain
  is.numeric(unlist(assignment)) || stop("Non-numeric argument: tasks sizes must be
  sum(unlist(assignment) <= 0) == 0 || stop("Invalid argument: tasks sizes must be

} # end function - .validate.assignment



#' Verify that assignment attributes are valid
#'
#' @param assignment Array of task sizes
#' @examples
#' a <- get.initial.assignment(2, c(10))
#' .validate.assignment.attributes(a)
#' attr(a, 'score') <- 0
#' attr(a, 'runtime95pct') <- 0
#' attr(a, 'runtime99pct') <- 0
#' .validate.assignment.attributes(a)
.validate.assignment.attributes <- function(assignment) {

  is.numeric(attr(assignment, 'score')) || stop("Invalid argument: assignment score
  attr(assignment, 'score') >= 0 || stop("Invalid argument: assignment score must b

  is.numeric(attr(assignment, 'deadline')) || stop("Invalid argument: assignment de
  attr(assignment, 'deadline') > 0 || stop("Invalid argument: deadline must be > 0"

  is.numeric(attr(assignment, 'runtime95pct')) || stop("Invalid argument: assignmen
  attr(assignment, 'runtime95pct') >= 0 || stop("Invalid argument: assignment runti

  is.numeric(attr(assignment, 'runtime99pct')) || stop("Invalid argument: assignmen
  attr(assignment, 'runtime99pct') >= 0 || stop("Invalid argument: assignment runti

} # end function - .validate.assignment



#' Verify that the assignment has the minimum number of tasks required
```

```
#'
#' @param assignment List mapping tasks to instances
#' @param min.num.tasks Minimum number of tasks in assignment
#' @examples
#' a <- get.initial.assignment(2, c(10))
#' .validate.num.tasks.in.assignment(a, 2)
#' .validate.num.tasks.in.assignment(a, 5)
.validate.num.tasks.in.assignment <- function(assignment, num.tasks.required) {

  num.tasks.available <- length(unlist(assignment))
  if (num.tasks.available >= num.tasks.required) {
    return (TRUE)
  } else {
    return (FALSE)
  } # end if - move more tasks than available?

} # end function .validate.num.tasks.in.assignment


#' Verify that runtimes are valid values
#'
#' @param runtimes Matrix of runtime of past runs for the given instance type. Each
#' @examples
#' r <- matrix(c(1,1), nrow=1, ncol=2)
#' .validate.runtimes.summary(r)
.validate.runtimes <- function(runtimes) {

  !missing(runtimes) || stop("Missing required argument: Must specify a numeric mat
  is.matrix(runtimes) || stop("Invalid argument type: Must specify a numeric matrix
  NCOL(runtimes) == 2 || stop("Invalid argument dimensions: Must specify a numeric
  NROW(runtimes) > 0 || stop("Invalid argument dimensions: Must specify a numeric m
  is.numeric(runtimes) || stop ("Invalid argument: Must specify a numeric matrix wi
  all(runtimes[,1] > 0) || stop("Invalid argument: 1st column (size) must have posi
  all(runtimes[,2] >= 0) || stop("Invalid argument: 2nd column (runtime) must have

} # end function - .validate.runtimes



#' Verify that runtime summaries are valid values
#'
#' @param runtimes.summary Numeric matrix containing mean and variance of runtimes
```

```
#' @examples

#' rs <- matrix(c(1,1,1), nrow=1, ncol=3)
#' .validate.runtimes.summary(rs)
.validate.runtimes.summary <- function(runtimes.summary) {

  !missing(runtimes.summary) || stop("Missing required argument: Must specify a nu
  is.matrix(runtimes.summary) || stop("Invalid argument type: Must specify a numeri
  NCOL(runtimes.summary) == 3 || stop("Invalid argument dimensions: Must specify a
  NROW(runtimes.summary) > 0 || stop("Invalid argument dimensions: Must specify a r
  is.numeric(runtimes.summary) || stop ("Invalid argument: Must specify a numeric m
  all(runtimes.summary[,1] > 0) || stop("Invalid argument: 1st column (size) must h
  all(runtimes.summary[,2] > 0) || stop("
Invalid argument: 2nd column (runtime) must have positive values")
  all(runtimes.summary[,3] >= 0) || stop("Invalid argument: 3rd column (var(runtime

} # end function - .validate.runtimes.summary



.validate.instance.type <- function(instance.type) {

  !missing(instance.type) || stop("Missing required argument: Must specify instance
  length(instance.type) != 0 || stop("Invalid argument length: instance.type must b
  nchar(instance.type) > 0 || stop("Invalid argument length: instance.type must be
  is.character(instance.type) || stop ("Invalid argument type: instance.type must b
  NROW(instance.type) == 1 || stop ("Invalid argument length: instance.type must be

} # end function - .validate.runtimes


# -----
# Other internal functions
# -----


#' Get runtimes for instance type
#'
#' @inheritParams setup.trainingset.runtimes
#' @param summary Return only summary of runtimes.
#' @return
#' If summary=F, return value is a matrix of runtimes for the given instance type.
```

```
#' If summary=T, return value is a matrix of summary of runtimes for the given inst
#' @examples
#' .get.trainingset.runtimes('m3xlarge')
.get.trainingset.runtimes <- function(instance.type, summary=F) {

  if (summary) {
    varname <- paste(instance.type, '.runtimes.summary', sep='')
  } else {
    varname <- paste(instance.type, '.runtimes', sep='')
  } # end if - get summary?

  exists(varname, envir=data.env) || stop("Runtimes for ", instance.type, " not set
  var <- get(varname, envir=data.env) # get var from internal env (data.env)
  return (var)

} # end function - .get.trainingset.runtimes



#' Get initial assignment of tasks to instances in a cluster
#'
#' Tasks are randomly assigned to instances
#'
#' @inheritParams get.initial.assignment
#' @return List containing a mapping of tasks to instances in cluster. The list inc
#' @examples
#' assignment <- get.initial.assignment.random(4, 1:30)
.get.initial.assignment.random <- function(cluster.size, task.sizes) {

  assignment <- vector('list', cluster.size)
  num.tasks <- length(task.sizes)
  idx.shuffle <- sample(num.tasks, replace=F)
  shuffled.task.sizes <- task.sizes[idx.shuffle]

  for (i in 1:num.tasks) {

    # get random instance
    inst <- sample(length(assignment), 1)
      assignment[[inst]] <- c(assignment[[inst]], shuffled.task.sizes[i])

  } # end for - loop over all tasks in order
```

```
  return (assignment)

} # end function - get.initial.assignment.random



#' Get initial assignment of tasks to instances in a cluster
#'
#' Tasks are assigned to instances in decreasing order of expected processing time
#'
#' @inheritParams get.initial.assignment
#' @return List containing a mapping of tasks to instances in cluster. The list inc
#' @examples
#' rs <- matrix(nrow=2, ncol=3)
#' rs[1,1] <- 10; rs[1,2] <- 23.5; rs[1,3] <- 2.5
#' rs[2,1] <- 20; rs[2,2] <- 33.5; rs[2,3] <- 3.5
#' assignment <- get.initial.assignment.leptf(2, rep(c(1,2), 3), rs)
.get.initial.assignment.leptf <- function(cluster.size, task.sizes, runtimes.summar

  assignment <- vector('list', cluster.size)
  total.runtimes <- array(0, dim=cluster.size) # to keep track of total runtimes in
  num.tasks <- length(task.sizes)

  means <- sapply(task.sizes, function(x) { idx <- which(runtimes.summary[,1] == x)
  size.means <- cbind(task.sizes, means)
  size.means <- size.means[order(size.means[,2], decreasing=TRUE), ]
  if (class(size.means) == 'numeric') size.means <- as.matrix(t(size.means))
  colnames(size.means) <- NULL
  rownames(size.means) <- NULL

  for (i in 1:num.tasks) {

      instance.with.smallest.total.runtime <- which.min(total.runtimes)
      # if multiple elements in list have the lowest value, which.min returns the f
      # for our purposes, it doesn't matter which of the instances with the lowest

      assignment[[instance.with.smallest.total.runtime]] <- c(assignment[[instance.
    total.runtimes[instance.with.smallest.total.runtime] <- total.runtimes[instance

  } # end for - loop over all tasks in order

  return (assignment)
```

```
} # end function - get.initial.assignment.leptf



#' Get list of instances that have the minimum number of tasks required
.get.admissable.instances <- function(assignment, num.tasks.per.instance, num.insta

  num.tasks.in.instances <- lapply(assignment, length)
  admissable.instances <- which(num.tasks.in.instances >= num.tasks.per.instance)
  return (admissable.instances)
} # end function - get.admissable.instances



#' Get number of instances depending on whether to exchange tasks or move tasks
.get.num.instances <- function(exchange) {
  num.instances <- 1
  if (exchange) num.instances <- 2

  return (num.instances)

} # end function - .get.num.instances



#' Get temperature for current iteration
#'
#' Temperature decreases linearly with each iteration
#'
#' @inheritParams get.temperature
#' @return Value of temperture for the current iteration (integer)
#' @examples
#' temp <- .get.temperature.linear.decrease(25, 100, 7)
.get.temperature.linear.decrease <- function(max.temp, max.iter, cur.iter) {

  # cur.iter is guaranteed to be at most 1 less than max.iter
  # so cur.temp will always be > 0
  cur.temp <- (max.iter-cur.iter)*(max.temp/max.iter)
  return (cur.temp)

} # end function - get.temperature.linear.decrease
```

```
#' Get bootstrap sample for a task in the input job
#'
#' @param input.size Task size for which samples are required (integer)
#' @param num.samples Number of samples required  (integer)
#' @param runtimes Matrix containing size & runtime info for training set samples
#' @return Matrix containing required number of samples for the given size
.bootstrap.get.task.sample <- function(input.size, num.samples, runtimes) {

  varname <- paste('runtimes.', input.size, sep='')
  runtimes.cur.size <- get(varname, envir=data.env)
  num.rows <- NROW(runtimes.cur.size)

  num.rows > 0 || stop('Cannot find any samples for size=', input.size, ' in traini

  idx <- sample(1:num.rows, num.samples, replace=T)
  s <- runtimes.cur.size[idx,]

  # transpose data frames due to the way they are 'flattened' in unlist
  if (N
ROW(s) > 1) s <- t(s)

  return (s)
} # end function - .bootstrap.get.task.sample


#' Get bootstrapped samples for all sizes in the input job
.bootstrap.get.job.sample <- function(size.reps.table, runtimes) {

  # FORMAT of size.reps.table (generated via aggregate())
  # > size.reps.table
      # Group.1 x
  # 1      10 1
  # 2      90 1
  # 3     200 1
  # 4     850 1
  # 5    2100 1

  samples.list <- apply(size.reps.table, 1, function(x) { .bootstrap.get.task.sampl
  samples.matrix <- matrix(unlist(samples.list), ncol=2, byrow=TRUE)
```

```
  return (samples.matrix)

} # end function - .bootstrap.get.job.sample


.bootstrap.get.job.runtime <- function(size.reps.table, runtimes) {

  samples.matrix <- .bootstrap.get.job.sample(size.reps.table, runtimes)
  s <- sum(samples.matrix[,2])
  return (s)

} # end function - .bootstrap.get.job.runtime


.bootstrap.get.job.runtime.dist <- function(size.reps.table, num.bootstrap.reps, ru

  job.runtime.dist <- array(dim=num.bootstrap.reps)
  for(i in 1:num.bootstrap.reps) {
      r <- .
bootstrap.get.job.runtime(size.reps.table, runtimes)
    job.runtime.dist[i] <- r
  } # end for - perform required number of iterations

  return (job.runtime.dist)

} # end function - .bootstrap.get.job.runtime.dist



# -----
# Exported functions
# -----


#' Setup runtimes for given instance type
#'
#' All instances in a cluster are assumed to be of the same type
#'
#' @param instance.type Instance type of cluster (string). All instances in the clu
#' @param runtimes Matrix of runtimes for the given instance type. Each row in the
#' @export
```

```
#' @examples
#' runtimes <- cbind(rep(c(1,2), each=5), c(rpois(5,5), rpois(5,10)))
#' setup.trainingset.runtimes('m3xlarge', runtimes)
setup.trainingset.runtimes <- function(instance.type, runtimes) {

  # Validate args
  .validate.instance.type(instance.type)
  .validate.runtimes(runtimes)

  # Save runtimes of individual trials to use in bootstrap sampling
  varname <- paste(instance.type, '.runtimes', sep='')
  assign(varname, runtimes, envir=data.env) # create new var in internal env (data.


  # save runtime summary
  m <- aggregate(runtimes[, 2], by=list(runtimes[, 1]), mean)
  v <- aggregate(runtimes[, 2], by=list(runtimes[, 1]), var)
  mv <- cbind(m[, 1], m[, 2], v[, 2])
  colnames(mv) <- c('size', 'mean', 'var')

  varname <- paste(instance.type, '.runtimes.summary', sep='')
  assign(varname, mv, envir=data.env) # create new var in internal env (data.env)


  # save runtimes for each size in a separate var
  uniq.sizes <- unique(runtimes[,1])
  for (s in uniq.sizes) {
    varname <- paste('runtimes.', s, sep='')
    ss <- subset(runtimes, runtimes[,1]==s)
    assign(varname, ss, envir=data.env)
  } # end for - loop over all sizes

} # end function - setup.trainingset.runtimes



#' Get initial assignment of jobs to instances in a cluster
#'
#' @param cluster.size Number of instances in the cluster (+ve integer)
#' @param task.sizes Array of task sizes (+ve reals)
#' @param runtimes.summary Numeric matrix containing mean and variance of runtimes
#' @param method Method to use to assign tasks to instances. Must be one of ('rando
```

```
#' @return List containing a mapping of tasks to instances in cluster. The list inc
#' @export
#' @examples
#' assignment <- get.initial.assignment(3, 1:30)
#' rs <- matrix(nrow=2, ncol=3)
#' rs[1,1] <- 10; rs[1,2] <- 23.5; rs[1,3] <- 2.5
#' rs[2,1] <- 20; rs[2,2] <- 33.5; rs[2,3] <- 3.5
#' assignment <- get.initial.assignment(3, c(rep(10, 3), rep(20, 3)), rs, method='l
get.initial.assignment <- function(cluster.size, task.sizes, runtimes.summary, meth

  # Validate args
  .check.if.positive.integer(cluster.size)
  .check.if.positive.real(task.sizes)

  if (method=='random') {
    assignment <- .get.initial.assignment.random(cluster.size, task.sizes)
  } else if (method=='leptf') {
    .validate.runtimes.summary(runtimes.summary)
    assignment <- .get.initial.assignment.leptf(cluster.size, task.sizes, runtimes.
  } else {
    stop('Invalid argument: ', method, ' is not a valid value for method')
  } # end if - method=random?

  return (assignment)

} # end function - get.initial.assignment



#' Generate a neighbor to an assignment
#'
#' The input assignment is modified in one of several different ways, including
#' \itemize{
#'  \item Move a task from 1 instance to another
#'  \item Exchange a task with another instance
#'  \item Move 2 tasks from 1 instance to another
#'  \item Exchange 2 tasks with another instance
#'  \item Move 2 tasks from an instance to 2 other instance
#'  \item Exchange 2 tasks with 2 other instances
#'  \item and so on...
#' }
#' Only the first 2 methods are currently implemented with an equal probability of
```

```r
#'
#' @param assignment A list representing a mapping of tasks to instances in a clust
#' @return A list representing the modified assignment of tasks to instances in the
#' @export
#' @examples
#' assignment <- get.initial.assignment(3, 1:30)
#' proposed.assignment <- get.neighbor(assignment)
get.neighbor <- function(assignment) {

  ex <- sample(c(TRUE, FALSE), 1)

  num.tasks.in.instances <- sapply(assignment, length)
  num.tasks.in.instances <- round(num.tasks.in.instances/3)
  num.tasks <- sample(max(num.tasks.in.instances), 1)

  if (ex) { cat('Exchange', num.tasks, 'tasks \n\n') }
  else { cat('Move', num.tasks, 'tasks \n\n') }

  neighbor <- move.tasks(assignment, num.tasks, exchange=ex)

  return (neighbor)

} # end function - get.neighbor



#' Generate neighbor by moving 1 task
#'
#' Randomly select 2 instances in the cluster. Randomly select a task from one of t
#'
#' @param assignment A list representing the assignment for which a neighbor is des
#' @param num.tasks Integer representing the number of tasks to be moved from 1 ins
#' @param exchange Exchange tasks between instances instead of moving them
#' @return A list representing the neighboring assignment
#' @export
#' @examples
#' assignment <- get.initial.assignment(3, 1:30)
#' neighbor <- move.tasks(assignment, 1)
#' neighbor <- move.tasks(assignment, 1, exchange=TRUE)
move.tasks <- function(assignment, num.tasks, exchange=FALSE) {

  # Validate args
```

```
  .validate.assignment(assignment)
  .check.if.positive.integer(num.tasks)


  # Need at least 2 instances in assignment to move or exchange tasks
  num.instances.in.assignment <- length(assignment)
  if (num.instances.in.assignment < 2) { return (assignment) }


  # Check if we have sufficient # tasks in the assignment (across all instances)
  if (exchange) {
    # Check if we have enough tasks to exchange
    valid <- .validate.num.tasks.in.assignment(assignment, 2*num.tasks)

    if (! valid) {
      # If not, check if we have enough tasks to move
      cat('WARN: Cannot exchange', num.tasks, ' tasks between 2 instances. Moving',
      exchange <- FALSE
      valid <- .validate.num.tasks.in.assignment(assignment, num.tasks)
      if (! valid) {
        # If not, fail
        stop("Invalid argument: Insufficient number of task to move")
      } # end if - insufficient # tasks to move
    } # end if - have enough tasks to exchange?

  } else {
    # Check if we have enough tasks to move
    valid <- .validate.num.tasks.in.assignment(assignment, num.tasks)
    if (! valid) {
      # If not, fail
      stop("Invalid argument: Insufficient number of tas
k to move")
    } # end if - insufficient # tasks to move

  } # end if - exchange tasks?


  # number of instances to use depends on whether we are moving tasks or exchanging
  # - exchange requires 2 instances; move requires 1 instance
  num.instances.to.use <- .get.num.instances(exchange)
```

```
  # Get all instances with at least num.tasks tasks
  all.admissable.instances <- .get.admissable.instances(assignment, num.tasks, num.

  # Can fail to get sufficient # admissable instances when:
  # exchange & # instances < 2
  # !exchange and # instances < 1 (due to insufficient # tasks to move in all insta

  if (  (exchange && (length(all.admissable.instances) < 2)) ||
        (length(all.admissable.instances) < 1) ) {
    # Insuffucient # admissable instances, so try moving 1 task between instances
    cat('WARN: Insufficient # instances to move/exchange tasks. Moving 1 task inste
    exchange <- F
    num.instances.to.use <- .get.num.instances(exchange) # use 1 instance
    num.tasks <- 1
    all.admissable.instances <- .get.admissable.instances(assignment, num.tasks, nu

    if(length(all.admissable.instances) < 1) {
      stop("Error: Cannot find a single instance with
at least 1 task!")
    } # end if - found at least 1 instance with 1 task?

  } # end if - sufficient # instances found?

  idx.admissable.instances.sample <- sample(1:length(all.admissable.instances), num
  admissable.instances.sample <- all.admissable.instances[idx.admissable.instances.

  # Remove task(s) from donor instance(s)
  tasks.mat <- matrix(nrow=num.instances.to.use, ncol=num.tasks)
  for (i in 1:num.instances.to.use) {

    inst <- admissable.instances.sample[i]
    num.tasks.in.instance <- length(assignment[[inst]])
    idx.tasks <- sample(1:num.tasks.in.instance, num.tasks)
    tasks <- assignment[[inst]][idx.tasks]

    assignment[[inst]] = assignment[[inst]][-idx.tasks]
    num.remaining.tasks.in.instance <- length(assignment[[inst]])
    if (num.remaining.tasks.in.instance == 0) assignment[inst] <- list(NULL)

    tasks.mat[i,] <- tasks
  } # end for - loop over all instances
```

```
  # TODO: need a more general way to do this
  if (exchange) {
    instance1 <- admissable.instances.sample[1]
    assignment[[instance1]] <- c(assignment[[instance1]], tasks.mat[2,])

    instance2 <- admissable.instances.sample[2]
    assignment[[instance2]] <- c(assignment[[instance2]], tasks.mat[1,])

  } else {
    # Get acceptor instance
    idx.remaining.instances <- (1:length(assignment))[-admissable.instances.sample]
    num.remaining.instances <- length(idx.remaining.instances)
    if (num.remaining.instances == 1) { instance2 <- idx.remaining.instances }
    else { instance2 <- sample(c(idx.remaining.instances), 1) }

    # Move the task to this instance
    assignment[[instance2]] <- c(assignment[[instance2]], tasks.mat[1,])

  } # end if - move only?

  attr(assignment, 'score') <- NULL
  attr(assignment, 'runtime95pct') <- NULL
  attr(assignment, 'runtime99pct') <- NULL

  return (assignment)

} # end sub - move.tasks



#' Compare 2 assignments based on their score
#'
#' Scores are calculated for both assignments. If the score of the proposed assignm
#'
#' @param cur.assignment Current assigment with score attribute (list)
#' @param proposed.assignment Proposed assignment with no score (list)
#' @param runtimes Matrix of runtimes for the given instance type. Each row in the
#' @param runtimes.summary Numeric matrix containing mean and variance of runtimes
#' @param deadline Time by which job must be complete (float). Same time units as r
#' @param max.temp Max temperature to use in the simulated annealing process (integ
#' @param max.iter Max # iterations to use to find the optimal assignment via simul
```

```
#' @param cur.iter Value of current iteration (integer)
#' @return A list containing the accepted assignment and score
#' @export
# @examples
# data('m3xlarge.runtimes.expdist')
# setup.trainingset.runtimes('m3xlarge', m3xlarge.runtimes.expdist)
# r <- get('m3xlarge.runtimes', envir=data.env)
# rs <- get('m3xlarge.runtimes.summary', envir=data.env)
# assign('runtimes.1', r, envir='data.env')
# c.a <- get.initial.assignment(2, c(1,1,1,1))
# c.a <- get.score(c.a, r, rs, 120)
# p.a <- get.neighbor(c.a)
# a <- compare.assignments(c.a, p.a, r, rs, 120, 25, 100, 7)
compare.assignments <- function(cur.assignment, proposed.assignment, runtimes, runt

  # Validate args
  .validate.assignment(cur.assignment)
  .validate.assignment.attributes(cur.assignment)
  .check.if.nonnegative.real(attr(cur.assignment, 'score'))
  .validate.assignment(proposed.assignment)

  .validate.runtimes(runtimes)
  .validate.runtimes.summary(runtimes.summary)

  .check.if.positive.real(deadline)
  length(deadline) == 1 || stop("Invalid argument length: deadline must be a single

  .check.if.positive.real(max.temp)
  length(max.temp) == 1 || stop("Invalid argument length: max.temp must be a single

  .check.if.positive.integer(max.iter)

  .check.if.nonnegative.integer(cur.iter)
  if (cur.iter >= max.iter) { stop('Invalid argument: cur.iter ', cur.iter, ' is >=

  proposed.assignment <- get.score(proposed.assignment, runtimes, runtimes.summary,

  cat('CURRENT.assignment: \n')
  print(cur.assignment)
  cat('\n')

  cat('PROPOSED.assignment \n')
```

```
  print(proposed.assignment)
  cat('\n')

  if (attr(proposed.assignment, 'score') >= attr(cur.assignment, 'score')) {
    cat('PROPOSED.score >= current.score. Returning PROPOSED \n\n')
    # new assignment has greater or equal prob. of completing job by deadline than
    result <- proposed.assignment

  } else {
    cat('proposed.score is lower \n')
      temp <- get.temperature(max.temp, max.iter, cur.iter)
      lhs <- round(exp((attr(proposed.assignment, 'score') - attr(cur.assignment,
      rhs <- round(runif (1, min=0, max=1), 2)
    cat('temp=',temp, ' lhs=',lhs, ' rhs=',rhs, '\n')

      if (lhs > rhs) {
      cat('lhs > rhs; returning PROPOSED \n\n')
      result <- proposed.assignment
      } else {
      cat('lhs <= rhs; returning CURRENT \n\n')
      result <- cur.assignment
      } # end if - lhs > rhs?

  } # end if - proposed.score >= cur.score?

  return (result)

} # end function - compare.assignments



#' Get score for input assignment
#'
#' @param assignment The assignment which needs to be scored (list)
#' @param runtimes Matrix of runtimes for the given instance type. Each row in the
#' @param runtimes.summary Numeric matrix containing mean and variance of runtimes
#' @param deadline Time by which job must complete (float; same units as runtimes)
#' @return The input assignment with a value for the score attribute. Score is the
#' @export
# @examples
# data('m3xlarge.runtimes.expdist')
# setup.trainingset.runtimes('m3xlarge', m3xlarge.runtimes.expdist)
```

```
# assignment <- get.initial.assignment(2, c(1,1,1,1))
# runtimes <- get('m3xlarge.runtimes', envir=data.env)
# runtimes.summary
 <- get('m3xlarge.runtimes.summary', envir=data.env)
# assignment <- get.score(assignment, runtimes, runtimes.summary, 60)
get.score <- function(assignment, runtimes, runtimes.summary, deadline) {

  # Validate args
  .validate.assignment(assignment)

  .validate.runtimes(runtimes)
  .validate.runtimes.summary(runtimes.summary)

  .check.if.positive.real(deadline)
  length(deadline) == 1 || stop("Invalid argument length: deadline must be a single

  num.instances <- length(assignment)
  scores <- matrix(nrow=num.instances, ncol=3)

  for (i in 1:num.instances) {

    tasks <- assignment[[i]]
    num.tasks <- length(tasks)

    if (num.tasks == 0) {
      scores[i] <- 1
      next;
    } # end if - any tasks on instance?

    g <- aggregate(tasks, by=list(tasks), FUN=length)

    if (num.tasks > bootstrap.threshold) {
      cat('Using Normal approx. to runtime dist. \n')
      means <- apply(g, 1,
                     function(x) { runtimes.summary[which(runtimes.summary[,1] ==
          )

          vars <- apply(g, 1,
                     function(x) { runtimes.summary[which(runtimes.summary[,1] ==
          )

          job.mean <- sum(means)
```

```
            job.sd <- sqrt(sum(vars))

          # score for this instance = Prob(tasks on this instance completing by deadlin
          scores[i,1] <- round(pnorm(deadline, mean=job.mean, sd=job.sd), 2)
          scores[i,2] <- round(qnorm(0.95, mean=job.mean, sd=job.sd), 2)
          s
cores[i,3] <- round(qnorm(0.99, mean=job.mean, sd=job.sd), 2)

        } else {
          cat('Using boostrap approx. to runtime dist. \n')
          bootstrap.dist <- .bootstrap.get.job.runtime.dist(g, num.bootstrap.reps, runt

          # Prob. of this instance completing by deadline
          ecdf.fn <- ecdf(bootstrap.dist)

          scores[i,1] <- round(ecdf.fn(deadline), 2)
          scores[i,2] <- round(quantile(bootstrap.dist, 0.95), 2)
          scores[i,3] <- round(quantile(bootstrap.dist, 0.99), 2)

        } # end if - more than bootstrap.threshold tasks?

      } # end for - loop over all instances in assignment

      # Return score & times of instance with least prob of completing by deadline
      min.idx <- which.min(scores[,1])

      attr(assignment, 'score') <- scores[min.idx, 1]
      attr(assignment, 'deadline') <- deadline
      attr(assignment, 'runtime95pct') <- scores[min.idx, 2]
      attr(assignment, 'runtime99pct') <- scores[min.idx, 3]

      return (assignment)

    } # end function - get.score



    #' Get temperature for current iteration
    #'
    #' @param max.temp Max value of temperature to use (float)
    #' @param max.iter Max number of iterations to search for optimal solution (integer
    #' @param cur.iter Value of current iteration (integer)
```

```
#' @param method Method used to decrease temperature. Currently only linear decreas
#' @return Value of temperture for the current iteration (integer)
#' @export
#' @examples
#' temp <- get.temperature(25, 100, 7)
get.temperature <- function(max.temp, max.iter, cur.iter, method='linear') {

  # Validate args
  .check.if.positive.real(max.temp)
  length(max.temp) == 1 || stop("Invalid argument length: max.temp must be a single
  .check.if.positive.integer(max.iter)
  .check.if.nonnegative.integer(cur.iter)
  if (cur.iter >= max.iter) { stop('Invalid argument: cur.iter ', cur.iter, ' is >=

  if (method=='linear') {
    temp <- .get.temperature.linear.decrease(max.temp, max.iter, cur.iter)
  } else {
    stop('Invalid argument: ', method, ' method of decreasing temperature is invali
  }# end if - linear decrease in temp?

  return (temp)

} # end function - get.temperature



#' Find optimal schedule
#'
#' Want an assignment with >= .95 probability of completing job by the deadline wit
#'
#' @param job Array of integers representing sizes of tasks in job
#' @param deadline Time (in seconds) by which job must be completed (integer)
#' @param cluster.instance.type Instance type of cluster (string). All instances in
#' @param cluster.size Integer representing the number of instances in the cluster
#' @param max.iter Max number of iterations to use to find the optimal assignment (
#' @param max.temp Max temperature to use in the simulated annealing process (flaot
#' @param reset.score.pct Begin next iteration from the best assignment if the diff
#' @param reset.num.iters Begin next iteration from the best assignment if the numb
#' @param debug Print debug info
#' @return A list representing the optimal assignment that could be found under the
#' @export
#' @examples
```

```
#' job <- c(1,60,100)
#' deadline <- 300
#' cluster.instance.type <- 'm3xlarge'
#' cluster.size <- 2
#' max.iter <- 2
#' max.temp <- 0.5
#' data(m3xlarge.runtimes.expdist)
#' setup.trainingset.runtimes('m3xlarge', m3xlarge.runtimes.expdist)
#' best.schedule <- schedule(job, deadline, cluster.instance.type, cluster.size, ma
schedule <- function(job, deadline, cluster.instance.type, cluster.size, max.iter,

  start.time <- proc.time()

  if (!is.null(reset.score.pct)) .check.if.positive.real(reset.score.pct)
  if (!is.null(reset.num.iters)) .check.if.positive.integer(reset.num.iters)

  if (debug) {
    output.prefix <- paste(cluster.size, '-inst-', length(job), '-tasks-', max.iter
    filename <- filename <- paste(output.prefix, '.output.txt', sep='')
    sink(filename)
  }

  runtimes <- .get.trainingset.runtimes(cluster.instance.type)
  runtimes.summary <- .get.trainingset.runtimes(cluster.instance.type, summary=T)

  cur.assignment <- get.initial.assignment(cluster.size, job)
  cur.assignment <- get.score(cur.assignment, runtimes, runtimes.summary, deadline)

  best.assignment <- cur.assignment
  best.score <- attr(best.assignment, 'score')
  if (debug) cat('best score=', best.score, '\n')

  if (debug) {
    scores.timeseries <- matrix(nrow=(max.iter)+1, ncol=7)
    colnames(scores.timeseries) <- c('Iter', paste('Acpt_', deadline, 's', sep=''),

    scores.timeseries[1,1] <- 1

    scores.timeseries[1,2] <- attr(cur.assignment, 'score')
    scores.timeseries[1,3] <- attr(cur.assignment, 'runtime95pct')
    scores.timeseries[1,4] <- attr(cur.assignment, 'runtime99pct')
```

```
    scores.timeseries[1,5] <- attr(best.assignment, 'score')
    scores.timeseries[1,6] <- attr(best.assignment, 'runtime95pct')
    scores.timeseries[1,7] <- attr(best.assignment, 'runtime99pct')

    filename.ts <- paste(output.prefix, '-scores-timeseries.csv', sep='')
    conn <- file(filename.ts, open='wt')
      writeLines('# Input Params', con=conn)
      writeLines(paste('# job.array = ', paste(job, collapse=';'), sep=''), con=con
      writeLines(paste('# num.jobs = ', length(job), sep=''), con=conn)
      writeLines(paste('# deadline = ', deadline, sep=''), con=conn)
      writeLines(paste('# cluster.instance.type = ', cluster.instance.type, sep='')
      writeLines(paste('# cluster.size = ', cluster.size, sep=''), con=conn)
      writeLines(paste('# max.iter = ', max.iter, sep=''), con=conn)
      writeLines(paste('# max.temp = ', max.temp, sep=''), con=conn)
      writeLines(paste('# reset.score.pct = ', ifelse(is.null(reset.score.pct), 'NU
      writeLines(paste('# reset.num.iters = ', ifelse(is.null(reset.num.iters), 'NU
      writeLines(paste('# debug = ', debug, sep=''), con=conn)

      write.table(t(scores.timeseries[1,]), file=conn, sep=',', quote=FALSE, row.na
      flush(conn)
  } # end if - debug?


  # go from 0 to 1 less than max.iter
  # so we s
tart at max temp and end just above 0 and avoid divide-by-zero errors
  for (i in 0:(max.iter-1)) {

    if (debug) cat('\n\n==========\nSA iter: ', i, ' (', (i+2), ') \n', '==========

      proposed.assignment <- get.neighbor(cur.assignment)
      cur.assignment <- compare.assignments(cur.assignment, proposed.assignment, ru
    cur.score <- attr(cur.assignment, 'score')

    # update best score, if necessary
    if (cur.score > best.score) {
      best.assignment <- cur.assignment
      best.score <- attr(best.assignment, 'score')
    } # end if - cur assignment better than best assignment so far?

    if (debug) cat('best score=', best.score, '\n')
```

```
    # restart from current best assignment if score of current assignment is too lo
    if (!is.null(reset.score.pct)) {
      if (best.score == 0) best.score = 0.0001
      d <- (best.score - cur.score)
      d.pct <- 100*d/best.score
      if (d.pct > reset.score.pct) {
        cur.assignment <- best.assignment
        if (debug) cat('Resetting current assignment to best assignment since d.pct
      } # end if - reset current assignment to best assignment
    } # end if - reset.score.pct defined?



    if (debug) {
      scores.timeseries[(i+2),1] <- (i+2)

      scores.timeseries[(i+2),2] <- attr(cur.assignment, 'score')
      scores.timeseries[(i+2),3] <- attr(cur.assignment, 'runtime95pct')
      scores.timeseries[(i+2),4] <- attr(cur.assignment, 'runtime99pct')

      scores.timeseries[(i+2),5] <- attr(best.assignment,
'score')
      scores.timeseries[(i+2),6] <- attr(best.assignment, 'runtime95pct')
      scores.timeseries[(i+2),7] <- attr(best.assignment, 'runtime99pct')

      write.table(t(scores.timeseries[(i+2),]), file=conn, sep=',', quote=FALSE, ro
      flush(conn)
    } # end if - debug?

  } # end for - loop over all iterations


  # sort task.sizes in each instance
  for (i in 1:length(best.assignment)) {
    best.assignment[[i]] <- sort(best.assignment[[i]], decreasing=TRUE)
  } # end for - loop over all instance

  cat('\nBest score: ', attr(best.assignment, 'score'), '\n')
  cat('Best assignment: \n')
  print(best.assignment)
  cat('\n\n')
```

```
  d <- proc.time()-start.time
  cat('Time taken: ', d[3], ' seconds')

  if (debug)  {
    sink()
    close(conn)
  } # end if - debug?

  return (best.assignment)

} # end function - schedule
```